NAG1-602
IN-61
64808CR
P.19

A RIGOROUS APPROACH TO SELF-CHECKING PROGRAMMING

Kien A. Hua and Jacob A. Abraham

Computer Systems Group
Coordinated Science Laboratory
University of Illinois
Urbana, IL 61801

Phone: (217) 344-8087

ABSTRACT

Self-checking programming has been shown to be an effective concurrent error detection technique. The reliability of a self-checking program however relies on the quality of its assertion statements. A self-checking program written without formal guidelines could provide a poor coverage of the errors. This paper presents a constructive technique for self-checking programming. We have defined a Structured Program Design Language (SPDL) suitable for self-checking software development. A set of formal rules, has also been developed, that allows the transformation of SPDL designs into self-checking designs to be done in a systematic manner.

Keywords: Concurrent error detection, self-checking programming, program design, assertion statements, transformation.

```
(NASA-CR-180657)  A RIGORCUS APPROACH TO             N87-26517
SELF-CHECKING PROGRAMMING  (Illincis Univ.)
19 p  Avail: NTIS HC  A02/MF  A01   CSCL 09B
                                                     Unclas
                                            G3/61    0064808
```

Acknowledgment: This research was supported in part by a contract from IBM Corporation and in part by the National Aeronautics and Space Administration (NASA) under contract NASA NAG 1-602.

1. INTRODUCTION

Off-line circuit testing is one of the widely used techniques to detect physical failures and to ensure that a system is defect-free. Unfortunately, the decrease in geometries has increased the possibility of transient errors in computing systems that are based on VLSI technology. Since these errors are usually nonrecurring and not reproducible, off-line testing (useful for permanent faults) will not reliably detect transients. Current trend is to include *Concurrent Error Detection* (CED) capability into the design of digital systems in order to detect errors concurrently during their normal operations.

Traditionally, systems with CED are implemented using self-checking circuits [1] and/or hardware duplication [2]. Since these techniques require hardware redundancy, they are usually very expensive. Recently, a new approach that utilizes a combination of hardware and software redundancy to accomplish concurrent error detection has been proposed [3, 4]. These techniques assume that malfunctions such as processor failures, bus faults will cause errors in the control flow of the program. These techniques partition the assembly language level instructions of the application program into branch-less blocks. At compile time, cyclic codes or signatures are generated for the instruction stream of each block. During the execution time, the same cyclic coded signatures are regenerated by a linear feedback shift register. Error detection is performed by comparing the run-time generated signatures with the ones precomputed by the compiler. These schemes are inexpensive and provide excellent control flow monitoring capability. Unfortunately, they do not perform well in case of errors due to data type faults [5].

Until automatic program verification becomes a reality, the reliability of today software systems mainly rely on software testing. Software testing however has been criticized as being inadequate for it usually does not reveal all the software bugs. "Program testing can be used to show the presence of bugs, but never to show their absence !" is a now-famous statement [6]. For safety-critical applications, concurrent error detection of errors due to software bugs is therefore as important as the detection of errors due to physical failures in the hardware. The CED techniques we have discussed so far do not detect errors due to software faults. Since a computing system is made up of software components and hardware components, a CED technique that can detect errors due to either hardware failures or programming bugs in the software would be very desirable.

An important characteristic of CED techniques is the level at which the checking mechanism is placed. Gate-level techniques such as those using error detecting coding [1] usually assume the single or double stuck-at fault model. As the geometric features of integrated circuits become smaller, physical defects can affect a local area of a circuit, and stuck-at fault models are therefore not satisfactory. Functional-level techniques such as those used in algorithm-based fault tolerance [7, 8, 9] assume a more general model which allows any single module in the system to be faulty. These techniques hence can detect errors due to a block of faulty logic that is local to any single module.

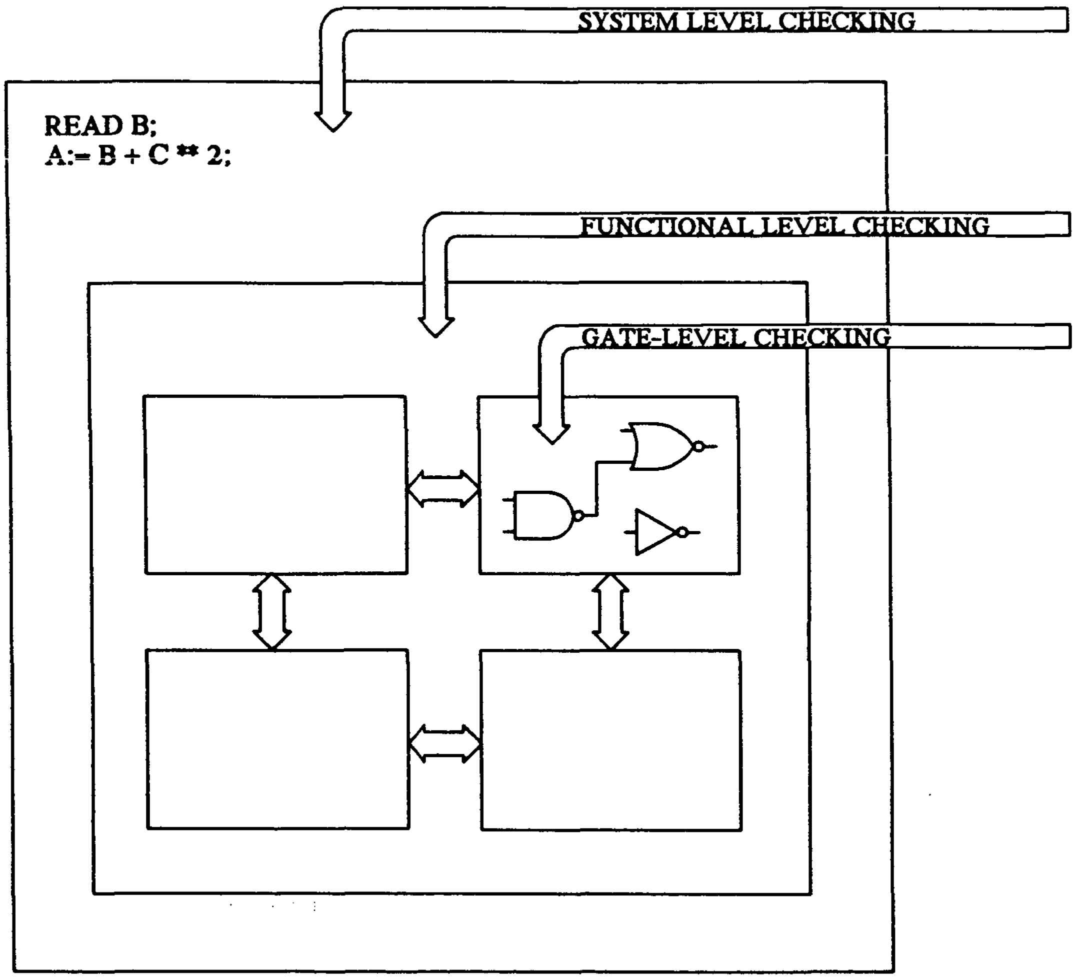

Figure 1 Three categories of concurrent error detection

In this paper we present a CED technique that places the checking mechanism at the application program level (Figure 1). This technique achieves error detection by introducing software redundancy in the form of executable assertions into the program to check the correct operation of the system during its execution. Both software and hardware faults that affect the dynamic behavior of the program therefore can be detected. Such computer programs that check their own dynamic behavior automatically during their execution are called self-checking programs, and we call the technique *Self-Checking Programming* (SCP). A more rigorous definition for self-checking program will be given in Section 3.

In recent years, several publications have proposed techniques for SCP [10, 11, 12, 13, 14]. In a previous paper [14] we also described a methodology for the experimental evaluation of the error coverage of self-checking programs. We also conducted experiments [14, 15] to study the effectiveness of SCP. The results showed that it can provide excellent coverage against both hardware and software faults.

In this paper we will present a constructive technique for SCP. A set of transformation rules that serves as mental aids in designing self-checking software will be given. The proposed method constructively extracts the assertions from the text of the program design. The correctness of the design is also verified based on the assertions, so that if all these assertions are TRUE when they are checked during run time, then the output computed must be correct with respect to the output specification.

The remainder of this paper is organized as follows: In Section 2, we define a *Structured Program Design Language* (SPDL) suitable for the design of self-checking software. The proposed technique for self-checking software design is presented in Section 3. In Section 4 the technique is extended to handle procedure and function calls. Finally, an overview of the SCP methodology is discussed in Section 5.

2. SPDL - A DESIGN LANGUAGE FOR STRUCTURED PROGRAMS

The properties of being intuitive and being rigorous often seem in conflict. A program design language that is too rigorous is usually too tedious to use, and it is error-prone. The algorithms it express also tends to be long and unintuitive. Its correctness is therefore more difficult to prove. Since the purpose of a design is to communicate the designer's idea to programmers, a good program design language should be readable. Yet it must be rigorous enough so that its correctness can be formally verified. Besides, a good program design language should also be able to present idea in varying degrees of detail, and the design structure must allow easy transformation of the program design into program code.

The SPDL we describe in this section is a language for the design of structured programs [16]. Its syntax is derived from Pascal. A SPDL assignment statement is an English statement that may contain mathematical and logical notations. For instance, the following statement is a SPDL assignment statement that computes the solution of a quadratic equation, and assigns the solutions to the variables X1 and X2:

$$(\text{X1}, \text{X2}) \leftarrow \text{Compute X such that } AX^2+BX+C = 0;$$

An advantage of using "pidgin" English statements is being able to express idea in varying degrees of details. Depending on the problem at hand, the algorithms can be expressed at whatever level of detail is appropriate to avoid ambiguity. For instance, the above SPDL statement can be rewritten in more details as follows:

$$\text{X1} \leftarrow (-B+\sqrt{B^2-4AC})/(2A)$$
$$\text{X2} \leftarrow (-B-\sqrt{B^2-4AC})/(2A)$$

the above two assignment statements provide two additional information: the formulas to compute the solutions, and the assignment of the solutions to the appropriate variables.

A program design written in SPDL has its statements embedded in the so called structured controls: ALTERNATIVE, ITERATIVE and SEQUENTIAL. IF..THEN..ELSE, WHILE..DO and SEQUENTIAL have been proposed as the building blocks for good programming [17]. Our experience has shown that IF..THEN and CASE (as in Pascal) are frequently useful in program design. We thus include all these five features in SPDL. If the reader accepts "other statements" as indicating, say, assignment statements and procedure calls, we can give the BNF [18] syntax description for the SPDL control constructs. In the following we have extended BNF with the convention that $(...)^+$ denotes "one or more instances of the enclosed."

```
<statement list> ::= (<statement>)+
<statement> ::= <alternative> | <iterative> | <other statements>
<alternative> ::= <ifthen> | <ifthenelse> | <case>
<ifthen> ::= if <boolean expression> then <statement list> fi;
<ifthenelse> ::= if <boolean expression>
                     then <statement list>
                     else <statement list> fi;
<case> ::= case <expression> of
                (<constant>: <statement list>)+
           endcase;
<iterative> ::= while <boolean expression> do
                     <statement list>
                enddo;
```

SPDL also supports hierarchical designs. The syntax for the procedure and function calls is the same as that of Pascal. So is the syntax for the declaration statements.

While the use of pidgin languages is also advocated by others, we have taken the additional steps of imposing a degree of formalism on the language so that the correctness of a design written in SPDL can be formally verified. This will be discussed in Section 3. The use of the structured controls also makes the transformation of a SPDL design into code in many structured programming languages (e.g., Pascal, PL/I, Fortran) a straightforward process.

3. DESIGN OF SELF-CHECKING SOFTWARE

In this section we assume that the design of a program has been successfully written in SPDL. We will present a constructive technique that transforms SPDL designs into self-checking designs.

Definition 1: An asserted program is said to be self-checking with respect to its output specification if any time it is run on some input data and all the assertions are TRUE when they are checked, then the output computed must be correct with respect to the output specification.

Definition 2: A program design is said to be self-checking if the program code derived from it is a self-checking program.

Definition 3: Let SL be a SPDL statement list. *ASSERT(SL)* denotes the assertion statements (knowledge) of what must be true after the "execution" of the SL.

Note that if p is a boolean expression, then ASSERT(p)=p.

Definition 4: Let SL be a SPDL statement list. *SCP(SL)* denotes the self-checking version of the SL.

Note that SCP(SL) denotes one version of the many possible self-checking versions for SL. In the following "SCP(P)=Q" should be read "Q is a self-checking version of P". It is not the only version.

In this paper we assume that the input data always satisfies the input specification of a program, otherwise the input specification could be used to check this property.

Theorem 1: Let P be a program (design). If ASSERT(P) implies the output specification, $\Psi(P)$, then the composition of P and $\Psi(P)$ is a self-checking program (design). In notation form, we have:

$$\forall P \quad ((\mathrm{ASSERT}(P) \rightarrow \Psi(P)) \rightarrow (\mathrm{SCP}(P)=(P\ ;\ \mathrm{ASSERT}(P))))$$

where the semicolon (;) denotes procedural composition.

Proof: The property (ASSERT(P) $\rightarrow$ $\Psi(P)$) guarantees that if the computed outputs satisfy ASSERT(P) (i.e., all the assertions in ASSERT(P) return TRUE), then they must also satisfy $\Psi(P)$. The computed outputs therefore must be correct with respect to the output specification of the program. From Definition 1, we thus have SCP(P)=(P ; ASSERT(P)).

□

Definition 5: Let A_1 and A_2 be assertion statements. $A_1 \bullet A_2$ denotes the new assertion statements that express the combined knowledge of what must be true derivable from A_1 and A_2.

Note that since the alternation of the order of the program statements may not preserve the semantics of the program, the operation $\bullet$ is not commutative. For instance, suppose we have the following assertions:

$$A_1 \equiv (X=a)$$
$$A_2 \equiv (X=X_0+2)$$
$$A_3 \equiv (Y=X+2)$$

where X_0 denotes the previous value of the variable X. We then have:

$$A_1 \bullet A_2 \bullet A_3 \equiv (X=a+2 \wedge Y=a+4)$$
$$A_1 \bullet A_3 \bullet A_2 \equiv (X=Y=a+2)$$

An assertion statement should have the form "*if not* ASSERTION *then* ERROR". However we will use the ASSERTION to denote the assertion statement whenever this is clear from the surrounding context to do so.

In order to be able to transform a SPDL design into a self-checking design, we need a set of transformation rules that transform SPDL control constructs into their self-checking versions. In the following we will use the symbols p to denote a boolean expression, k to denote an expression, SLi to denote statement list i and α_i to denote constant i.

(1) **Rule of Composition (Sequential):**

ASSERT(SL1 SL2) = ASSERT(SL1) • ASSERT(SL2)

SCP(SL1 SL2) = (SL1 SL2 ASSERT(SL1 SL2))

(2) **Rule of Alternation:**

(a) **Rule of ifthen:**

ASSERT(*if* p *then* SL *fi*;) = (p $\wedge$ ASSERT(SL)) $\vee$ $\neg$p

SCP(<ifthen>) = (<ifthen> ASSERT(<ifthen>))

(b) **Rule of ifthenelse:**

ASSERT(*if* p *then* SL1 *else* SL2 *fi*;)

= (p $\wedge$ ASSERT(SL1)) $\vee$ ($\neg$p $\wedge$ ASSERT(SL2))

SCP(<ifthenelse>) = (<ifthenelse> ASSERT(<ifthenelse>))

(c) **Rule of Case:**

ASSERT(*case* k *of* α_1: SL1 ... α_n : SLn *endcase;*)

= ((k=α_1) $\wedge$ ASSERT(SL1)) $\vee$ ... $\vee$ ((k=α_n) $\wedge$ ASSERT(SLn))

SCP(<case>) = (<case> ASSERT(<case>))

(3) **Rule of Iteration:**

SCP(*while* p *do* SL *enddo*;)

= (*while* p *do* ASSERT(p) SL ASSERT(SL) *enddo*; ASSERT(¬p))

The assertions as described in Theorem 1 are called global assertions. In some cases, it is more convenient and more efficient to partition a program (or procedure) into segments, and local assertions are inserted at the end of each segment to make them self-checking.

Theorem 2: Let P be a program (design) with embedded local assertions such that each segment is self-checking. If the set of local assertions implies the output specification, $\Psi(P)$, of the program (design), then P is self-checking with respect to the output specification. In notation form, we have:

$$\forall\ P,\ P=(\mathrm{SCP(SL1)\ SCP(SL2)\ \ldots\ SCP(SLn)})$$
$$((\mathrm{ASSERT(SL1)} \cdot \mathrm{ASSERT(SL2)} \ldots \cdot \mathrm{ASSERT(SLn)} \rightarrow \Psi(P)) \rightarrow \mathrm{SCP}(P)=P)$$

where SLi denotes i-th segment of the program (design) P.

Proof:

P: SCP(SL1) SCP(SL2) ... SCP(SLn) = SL1 ASSERT(SL1) ... SLn ASSERT(SLn)

Apply the Rule of Composition, P can be written as:

P': SL1 SL2 ... SLn (ASSERT(SL1) • ASSERT(SL2) • ... • ASSERT(SLn))

Since (ASSERT(SL1) • ... • ASSERT(SLn))→Ψ(P), based on Theorem 1 we can conclude that P' is a self-checking program (design). P is however semantically equivalent to P', hence P is also a self-checking program (design).

□

The Rule of Iteration is based on Theorem 2. A WHILE..DO statement can be considered as a sequence of IF..THEN statements:

(*while* p *do* SL *enddo*;) ≡ (*if* p *then* SL *fi*;
while p *do* SL *enddo*;)
≡ (*if* p *then* SL *fi*;
.
.

if p *then* SL *fi*;)

According to Theorem 2, we can break this statement list into individual IF..THEN statements and insert assertions for each of them. Since ASSERT(SL) is the same for each of the IF..THEN statements, the same ASSERT(SL) can be used to check the correct behavior of each iteration of the WHILE..DO statement. Since we may not know the exact number of iterations, induction might be required to prove the correctness of a design that contains iterative statements.

Example 1: This example illustrates the application of the transformation rules. The following procedure, MATRIXINV, computes the inverse of the n×n matrix M, which is returned as MI.

```
procedure MATRIXINV(n: integer;  (* dimension of M *)
                    M: array [1..10,1..10] of real;  (* matrix to invert *)
                    var MI: array [1..10,1..10] of real;  (* inverted matrix *)
var
   i: integer;  (* loop index *)
   B,X: array [1..10] of real;
begin
1  i←1;
   while i≤n do
2       B←i-th column of the identity matrix, I[i-th column];
3       MI[i-th column]←compute X such that MX=B;
4       i←increment i;
   enddo
end;  (* MATRIXINV *)
```

The assignment statements in the procedure MATRIXINV have been numbered. In the following discussion, we will refer to the statements by their numbers. Also, i_0 denotes the previous value of the variable i.

SCP(MATRIXINV) = SCP(1 *while* i$\leq$n *do* 2 3 4 *enddo*;)

Apply Theorem 2, we have:

SCP(MATRIXINV) = (SCP(1) SCP(*while* i$\leq$n *do* 2 3 4 *enddo*;))   (1)

Apply Theorem 1, we have:

```
SCP(1) = ( 1 ASSERT(1) )
       = ( 1 if i≠1 then ERROR; )      (2)
```

Apply the Rule of Iteration, we have:

```
SCP(while i≤n do 2 3 4 enddo;)
    = ( while i≤n do
          ASSERT(i≤n)
          2 3 4
          ASSERT(2 3 4)
        enddo
        ASSERT(i>n) )      (3)
```

Apply the Rule of Composition, we have:

$$\begin{aligned}\text{ASSERT}(2\ 3\ 4) &= \text{ASSERT}(2) \bullet \text{ASSERT}(3) \bullet \text{ASSERT}(4)\\ &= (B{=}I[\text{i-th column}]) \bullet (M{\cdot}MI[\text{i-th column}]{=}B) \bullet (i{=}i_0{+}1)\\ &= (M{\cdot}MI[\text{i-th column}]{=}I[\text{i-th column}]) \bullet (i{=}i_0{+}1)\\ &= (M{\cdot}MI[\text{i-th column}]{=}I[\text{i-th column}]) \wedge (i{=}i_0{+}1)\end{aligned}$$

We thus have:

```
SCP(while i≤n do 2 3 4 enddo;)
    = ( while i≤n do
            if i>n then ERROR
            $i_0$ ← i;   (* save previous value *)
            2 3 4
            if (M·MI[i-th column] ≠ I[i-th column]) then ERROR;
            if (i≠$i_0$+1) then ERROR;
        enddo
        if (i≤n) then ERROR; )      (4)
```

Substitute (2) and (4) into (1), we have:

```
SCP(MATRIXINV) = ( i ← 1;
                   if i≠1 then ERROR;
while $i \leq n$ do
    if $i > n$ then ERROR
    $i_0 \leftarrow i$;   (* save previous value *)
    B$\leftarrow$ I[i-th column];
    MI[i-th column] $\leftarrow$ compute X such that MX=B;
    i $\leftarrow$ increment i;
    if (M·MI[i-th column] $\neq$ I[i-th column]) then ERROR;
    if ($i \neq i_0+1$) then ERROR;
enddo
if ($i \leq n$) then ERROR; )
```

The assertions in (4) guarantee that after the execution of the WHILE..DO construct, we must have:

$$\forall i \leq n,\ M\cdot MI[i\text{-th column}]=I[i\text{-th column}]$$

However, the assertion ASSERT(1) ensures that i was initialized to "1" before entering the loop. We therefore have:

$$\forall i\ 1 \leq i \leq n,\ M\cdot MI[i\text{-th column}]=I[i\text{-th column}]$$

which is the output specification of the procedure MATRIXINV. From Theorem 2, we hence can conclude that the SCP(MATRIXINV) is self-checking with respect to the output specification. By doing the above reasoning, we have also proved the correctness of the design. We have not discussed the termination problems. A technique based on the well-found set properties described in [14] can be used to detect this class of errors, or we may design a computing system that includes a time-out mechanism for the detection of infinite loops [13].

The transformation process described in Example 1 may seem to involve many steps. Actually many of them were shown for the purpose of illustration. In practice, the obvious steps could have been skipped, and the process would have appeared much shorter.

4. HANDLE PROCEDURE CALLS

A SPDL design may involve procedure and function calls. In order to show the generality of the transformation technique, it is extended to handle procedure and function calls. There are two approaches to these statements:

(1) We can consider a procedure or a function as a separate program and apply the transformation rules to transform it into a self-checking design with respect to its own specification. The output specification of the called procedure or function then can be used as a lemma in the proof of self-checking for the calling procedure. If we consider the called procedure as a segment of the calling procedure, then Theorem 2 states that the calling procedure is self-checking if the output specification of the called procedure together with the ~~assertions of the calling procedure imply the output specification of the calling procedure.~~

(2) Viewing a procedure call as an instance of the called procedure, we can consider the called procedure as if it is part of the calling procedure. The self-checking transformation then can be carried out, according to the following rules, otherwise normally:

(a) At the beginning of the called procedure, we add assignment statements to assign the values of the actual parameters to the corresponding formal parameters.

(b) At the end of the called procedure, we add assignment statements to assign the values of those formal parameters that are "called by reference" to the corresponding actual parameters.

(c) Replace the procedure call statement by the new body of the called procedure.

For function calls, the rules are similar. They are given belows:

(a) At the beginning of the called function, we add assignment statements to assign the values of the actual parameters to the corresponding formal parameters.

(b) Add before the statement that involves the function call the new body of the called function.

(c) Replace the function call by the function name (i.e., the function name is considered as a variable).

The above rules describe merely a mental process. No actual modification to the original design should be done.

Example 2: This example illustrates these two approaches. The procedure MATRIXINV shown in Example 1 can be rewritten to include a procedure call to procedure SOLVE that computes the solution of a linear system MX=B. The solution is returned in the matrix X.

```
procedure MATRIXINV(n: integer;  (* dimension of M *)
                    M: array [1..10,1..10] of real;  (* matrix to invert *)
                    var MI: array [1..10,1..10] of real;  (* inverted matrix *)
var
    i: integer;  (* loop index *)
    B,X: array [1..10] of real;
begin
    i←1;
    while i≤n do
        B←i-th column of the identity matrix, I[i-th column];
        call SOLVE(M,X,B);
        MI[i-th column]← X;
        i←increment i;
    enddo
end;  (* MATRIXINV *)
```

If the first approach is used , then the SCP(MATRIXINV) is as follows:

```
procedure MATRIXINV(n: integer;  (* dimension of M *)
                    M: array [1..10,1..10] of real;  (* matrix to invert *)
                    var MI: array [1..10,1..10] of real;  (* inverted matrix *)
var
    i: integer;  (* loop index *)
    B,X: array [1..10] of real;
begin
    i←1;
    if (i≠1) then ERROR;
    while i≤n do
```

```
        if i>n then ERROR;
        $i_0$ ← i;  (* save previous value *)
        B←i-th column of the identity matrix, I[i-th column];
        call SOLVE(M,X,B);
        MI[i-th column]← X;
        i←increment i;
        if (MI[i-th column]≠x) then ERROR;
        if (i≠$i_0$+1) then ERROR;
    enddo
end;  (* MATRIXINV *)
```

Note that the self-checking properties of the procedure SOLVE is assumed by the calling procedure. MATRIXINV assumes that the solution returned in X has been checked to be correct by SOLVE, and it only checks to make sure that X is assigned to the appropriate column of the matrix MI. If the second approach was used, the assertion statements for the SCP(MATRIXINV) would have been the same as those shown in Example 1. Consider the procedure SOLVE as having only one statement:

X ← compute X such that MX=B;

It is then obvious why the second approach would have yield the same assertion statements as those derived in Example 1.

5. SELF-CHECKING PROGRAMMING

The traditional software development life cycle is as shown in Figure 2.a. Figure 2.b depicts the life cycle of the proposed SCP technique. In Figure 2.b, an additional stage is added to perform the self-checking transformation. A set of rules that guides the transformation process has been described in Section 3. Once the self-checking design is available, the next step is to translate it into code in the target language. Even though this translation process may introduce coding faults, the errors will be detected during run time because the design has already been proved to be correct and the assertion were extracted from the design text, not the program code. In other words, the

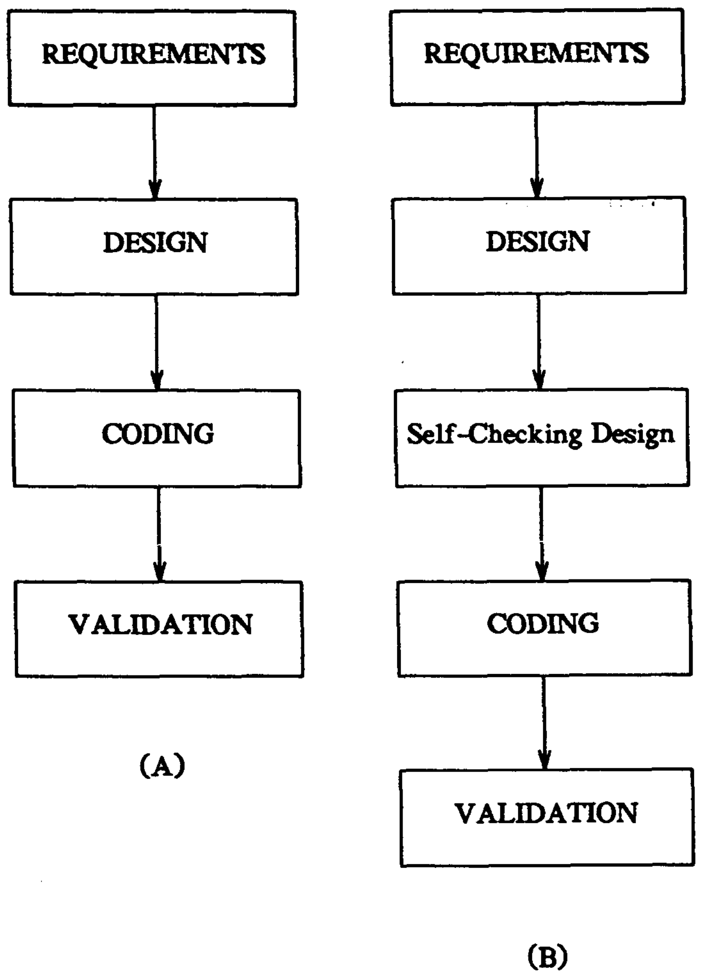

Figure 2. Traditional software development process (a) versus self-checking software development process (b).

assertions, derived in this manner, check whether the run time computation correctly computes the intended semantics defined by the program design. Any discrepancy caused by coding faults therefore will be detected.

The self-checking transformation process also proves the correctness of the design. As the software systems becoming more complex, it is advantageous to do as much validation as early as possible in the life cycle. Fixing a bug in the program design is obviously much less expensive then having to fix it in the program code. The necessity of proving the correctness of the program designs has become increasingly apparent.

If we consider the validation of the program design as a mandatory step in software development, then the only time overhead to make a program self-checking is the minimal time required to translate the assertion statements into code in the target language. This drawback however is offset by the fact that the assertion statements can detect and help the programmer to locate coding bugs. The debugging time therefore can be improved.

The CED technique we have described depends for its reliability improvement on the assumption that errors occur during the execution of the assertion statements will not mask out the errors in the normal computation. This is a reasonable assumption. An assertion statement usually has the following form:

if expression1 $\neq$ expression2 *then* ERROR

It is unlikely that errors occurred during the computation of expression1 and/or expression2, yet their results are accidentally equal. Our experimental results [14, 15] confirmed the validity of this assumption.

6. CONCLUSIONS

Self-checking programming has been shown to be an effective concurrent error detection technique [14, 15]. Its most distinguishing feature is the ability to detect all three classes of computing errors: errors due to hardware faults, transient faults or software faults. The reliability of a self-checking program however relies on the quality of its assertion statements. A self-checking program written without formal guidelines could provide a poor coverage of the errors.

This paper presents a constructive technique for self-checking programming. We have defined a structured program design language that is suitable for self-checking software development. A set of formal rules, has also been proposed, that makes the transformation of SPDL designs into self-checking designs a systematic process.

A disadvantage of self-checking programming is the extra time required to execute the assertion statements. A 1000-line self-checking Pascal program was implemented for an experimental study [15]. We found the performance overhead for this program to be 9.8%. If this is a typical figure, then it is negligible for most applications. Multiprocessor systems can also be used to improve the performance of self-checking programs. This subject is discussed in [14].

REFERENCES

[1] J. Wakerly, *Error Detecting Codes, Self-Checking Circuits and Applications*, New York: North-Holland, 1978

[2] D. Johnson, "The Intel 432: A VLSI Architecture for Fault-Tolerant Computer Systems," *IEEE Computer*, pp. 40-48, August 1984.

[3] T. Sridhar and S. M. Thatte, "Concurrent Checking of Program Flow in VLSI Processors," *Proc. Int'l Test Conf.*, pp. 191-199, 1982.

[4] J. P. Shen and M. A. Schuette, "On-Line Self-Monitoring Using Signatured Instruction Streams," *Proc. Int'l Test Conf.*, pp. 275-282, 1983.

[5] M. E. Schuette, J. P. Shen, D. P. Siewiorek and Y. X. Zhu, "Experimental Evaluation of Two Concurrent Error Detection Schemes," *Proc. 16th FTCS*, pp. 138-143, June 1986.

[6] E. W. Dijkstra, "Notes on Structured Programming," in O. J. Dahl, E. W. Dijkstra and C. A. R. Hoare, *Structured Programming*, New York: Academic Press, pp 1-82, 1972.

[7] K-H Huang and J. A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Trans. on Comput.*, vol. C-33, pp. 518-528, June 1984.

[8] J-Y Jou and J. A. Abraham, "Fault-Tolerant Matrix Arithmetic and Signal Processing on Highly Concurrent Computing Structures," *Proceedings of the IEEE Special Issue on Fault Tolerance in VLSI*, vol. 74, no. 5, pp. 732-741, May 1986.

[9] P. Banerjee and J. A. Abraham, "Fault-Secure Algorithms for Multiple-Processor Systems," *Proc. IEEE Int'l Symp. on Computer Architecture*, pp. 279-287, June 1984.

[10] S. S. Yau and R. C. Cheung, "Design of Self-Checking Software," *Proc. Int'l Conf. on Reliability Software*, pp. 405-457, April 1975.

[11] D. M. Andrews, "Using Executable Assertions for Testing and Fault Tolerance," *Proc. 9th Int'l Symp. on Fault Tolerant Computing*, pp. 102-105, June 1979.

[12] A. Mahmood, E. J. McCluskey and D. J. Lu, "Concurrent Fault Detection Using A Watchdog Processor and Assertions," *Proc. Int'l Test Conf.*, pp. 622-628, 1983.

[13] A. Milli, Self-Checking Programs: An Axiomatic Approach to The Validation of Programs by The Use of Assertions, PH. D. dissertation, Dept. Comp. Sci., University of Illinois, Urbana, 1981.

[14] K. A. Hua and J. A. Abraham, "Design of Systems with Concurrent Error Detection Using Software Redundancy," *Proc. FJCC '86*, pp. 826-835, Nov. 1986.

[15] K. A. Hua and J. A. Abraham, "Experimental Evaluation of Self-Checking Programming," submitted to the *17th FTCS*, June 1987.

[16] D. J. Dahl, E. W. Dijkstra and C. A. R. Hoare, *Structured Programming*, New York: Academic Press, Inc., 1972.

[17] E. W. Dijkstra, "Go To Statement Considered Harmful," *Comm. ACM*, vol. 11, pp. 147-148, Mar. 1968.

[18] Naur, Peter (Ed.), "Report on The Algorithmic Language ALGOL 60," *Comm. ACM*, vol. 3, pp. 299-314, May 1960.